Façade and the DP Wrap-Up
Which pattern does this class diagram from the Factory chapter call out for?

A. Strategy  
B. Decorator  
C. Adapter  
D. Factory
B: Decorator

We see a combinatorial explosion of options

There is a “plain” thing and “decorations” of it

Nearly identical to coffee drink example

EXERCISE:

Code up Pizza Decorator
abstract class Pizza {
  String description = null;
  String getDesc() { return description; }
...
}

class NYPizza extends Pizza {
  NYPizza () {
    description = “New York Pizza”;
  }
...
}
class ChicagoPizza extends Pizza { ...
}

abstract class PizzaDecorator extends Pizza {
  abstract String getDesc();
...
}
class Cheese extends PizzaDecorator {
  Pizza pizza;
  String getDesc() {
    return pizza.getDesc() + “cheese”;
  }
...
}

class Pepperoni extends PizzaDecorator {
  ...
}
Façade

It’s all a...oh, nevermind.
Real-world examples:
- Database connectors
- Chat systems

Adapter:
- Uses **Composition** and **Delegation**
- Adapter (wrapper) is **not subclass of wrapped object** (because we’re adapting)
- Naming: InputStreamToReaderAdapter
Façade – Variant of Adapter

- Not for matching incompatible interfaces, but for *simplifying use*
  - Hide unneeded operations
  - Combine low-level operations into convenient one
  - Combine several objects into one object

- Android’s AnimationUtils is an example

- Façades can be used to achieve a property called *layering* (CSE 120 Operating Systems)
Façade - Example

- Composes two or more classes
- Groups low-level operations into simpler high-level ones
- Façade doesn't have to be interface (but if it is, it will be adapter, too)
- If low-level classes declared as interfaces, then becomes Strategy, too

- Has some qualities of Mediator, as it coordinates sub-elements
- But also acts as go-between for clients, not just sub-elements
Which pattern?

Suppose we extended myCity to provide a graph of friend proximity over the whole day. We’d start with a simple display, but think we’ll eventually adopt a powerful library to support a glanceable, informative display. Which pattern should be applied?

A. Strategy
B. Decorator
C. Adapter
D. Façade
E. Factory
Discussion

- **Adapter**: graphing library probably incompatible with initial service
- **Façade**: to hide complexities of the powerful libraries
- **Decorator**: to add border, extra info, around graph view
- **Strategy**: to give users choice of graph
  - Qualities of strategy pattern in Adapter also facilitate testing
- **Factory**: making variants of display for different modes/views
- **Do all?** Agile says wait-and-see (refactor), as DPs take time and add complexity

Still, should compose abstract types (interfaces and abstract superclasses) in initial design

- almost free
- will save writing new testcases
- sets the stage for quick use of DPs later
Mocking: A Design Pattern Perspective on Testing

Yup, composing an interface and delegating to it.
Use Strategy to Fake A Resource

- External resources, sensors, etc. a problem in test
  - changing state, unrepeatable
  - sometimes inaccessible (GPS, network DB)
- Make sure such resources accessed through Strategy or Adapter
- Then implement “mock” alternatives for test to “fake” the resource

Here’s a strategy that will make good gift cards...

Here’s one that will make a gift card with insufficient funds on it.

Here’s the real MySQL accessor.

Here’s one that will throw an error regardless of what ID you give it.
public class TestOrderProcessing {
   // other tests

   public class TestAccessor implements DBAccessor {
      public GiftCard getGC(int gcId) {
         GiftCard gc = new GiftCard();
         gc.setActivationDate(new Date());
         gc.setExpirationDate(new Date());
         gc.setBalance(new BigDecimal(100));
      }
      // ... the other DBAccessor methods go here...
   }

   @Test
   public void testSimpleOrder() {
      // First create the order processor
      OrderProcessor orderProcessor = new OrderProcessor();
      orderProcessor.setDBAccessor(new TestAccessor());

      // Then we need to describe the order we're about to place
      OrderInformation orderInfo = new OrderInformation();
      orderInfo.setCustomerName("Dan");
      orderInfo.setDrinkDescription("Bold with room");
      orderInfo.setGiftCardNumber(12345);
      orderInfo.setPreferredStoreNumber(123);

      // Hand it off to the order processor and check the receipt
      Receipt receipt = orderProcessor.processOrder(orderInfo);
      assertNotNull(receipt.getPickUpTime());
      assertTrue(receipt.getConfirmationNumber() > 0);
      assertEquals(receipt.getGCBalance().equals(0));
   }
}
How to link Place-its and Marker?

- I decide want to have an Android map Marker for every Place-it

- Idea: subclass Marker so that Place-it is a Marker!

```java
class Place-it extends Marker {
    // new Place-it functionality
}
```

- What’s wrong with this proposed design?

A. Marker is “final”

B. Not OCP

C. Makes no sense for MS2, because category Place-its don’t go on map

D. Extending a concrete class

E. All of the above
So how do we link Place-it to Marker?

- class Place-it extends Marker {
  // new Place-it functionality
}

A. Observer (Listener)
B. Adapter
C. MVC (Mediator + 2 Observer)
D. Decorator
Marker is a **View** of Place-it

- Changes to Place-it need to be propagated to Marker: e.g., *delete*
- Implies Observer

But view is not passive; an active UI:

- Queries to Marker need to be propagated to Place-it (and back): *show description*
- Means that Place-it needs to “Observe” Marker, too

But don’t want Place-it & Marker to:

- be “coupled” to each other
- *know about* each other
- Marker is final, anyway!

So we introduce a Mediator to listen to both of them
MVC for Place-it / Marker

- Placeit
  - List<PlaceitListener> listeners

- PlaceitMarkerMediator
  - Map<Placeit,Marker> p2m
  - Map<Marker,Placeit> m2p
  - GoogleMap GMap

- PlaceitSubject
  - register, notify, ...

- PlaceitListener
  - onDelete()

- InfoWindowAdapter
  - getInfoContents(Marker)
  - getInfoWindow(Marker)

- GoogleMap
  - ...

- Marker
  - ...

<<interface>>

* PlaceitMarkerMediator

___ ____________________________
...
MVC Object Interactions

placeit.getDescription()
placeit.registerObserver(this)

marker.remove()  
gm.setInfoWindowAdapter(…)

onDelete(), etc.

Place-its

Markers

a GoogleMap

a PlaceitMarkerMediator
Take-Aways

- Design patterns are a vocabulary of design experience
  - Map a design problem to a design solution
  - Promote flexibility, reuse, enhancement, even testing
  - Several can be combined to solve complex problems
  - A short-cut to experience; a short-hand in conversation

- OCP and related principles are the theoretical underpinnings of DPs
  - Compose abstractions and delegate to them
  - When we see concrete referring to concrete in DP, it is to stop the hemorrhaging (e.g., Factory and Adapter)

- Both are widely used in Java & Android frameworks